Introduction To Apache Beam

Java MN User Group

Thank You!

Why Beam? Why Now?

- Past Java MN presentations have touched on increasingly popular and important aspects of handling large volumes of data at scale, in addition to how we incorporate traditional tenants of software engineering into the frameworks that support these technologies.
- We've discussed cloud computing in the context of AWS Lambdas, the fundamentals of machine learning models, and the incorporation of software design patterns into already highly abstracted frameworks.
- Apache Beam focuses on the *transmission* and *manipulation* of data for broader consumption. It can be hosted in cloud computing environments (most notably Google Cloud Platform), and is especially useful in the data collection and preparation phases of the machine learning life-cycle.

Layout

- Terminology
- History
- Implementation
- Resources

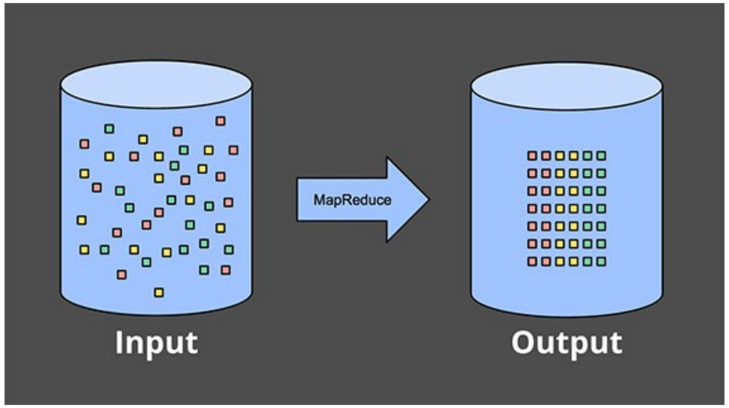
Apache Beam Defined

- *Apache Beam is an open source, unified model for defining both batch and streaming data-parallel processing pipelines.* *https://beam.apache.org/get-started/beam-overview/*

- Beam: Batch + Stream Processing

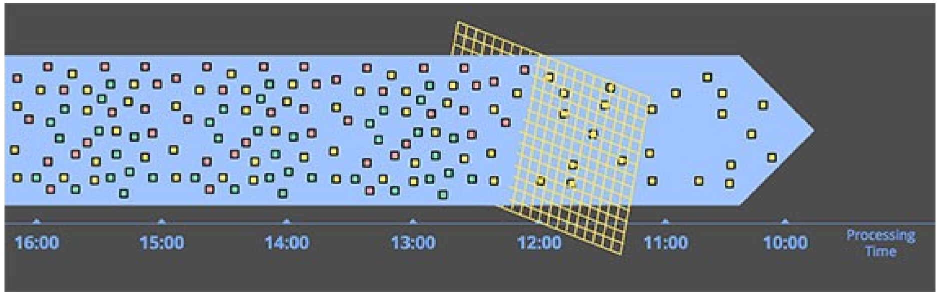
Batch

- Batch jobs, batch processing, or batching, reflect that idea that some work is being performed on a complete, whole, or bounded dataset. No new data enters the set under processing while the batch job is being run, or if it does, it is ignored.
- Common examples of batch processing include log file analysis, repeated runs of a stored procedure against an RDBMS, or analyzing the contents of a distributed dataset such as that within the HDFS (Hadoop File System).
- (Akidau, https://www.oreilly.com/ideas/the-world-beyond-batch-streaming-101)



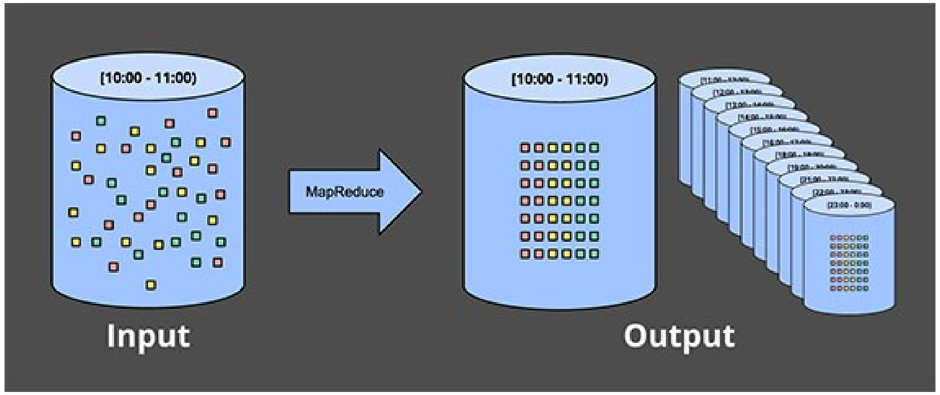
Streaming

- Unlike batch processing, stream processing is concerned with operating on unbounded datasets. These are collections of data that are constantly being updated, and have no defined ending.
- Messages being written to a bus like IBM MQ, RabbitMq, Apache Kafka, or Google Cloud PubSub is a great example of an unbounded dataset that is continually changing as new events are added to and consumed from a topic, subscription, or queue.
- (Akidau, https://www.oreilly.com/ideas/the-world-beyond-batch-streaming-101)



Batch Engines and Streaming Data

- Successive runs of batch processing jobs have been used for years to operate on streaming datasets, most commonly, by segmenting the data into fixed windows of time to create bounded datasets to operate on.
- However, the batch model has some inheritent shortcomings that Apache Beam seeks to address. We'll talk about these next.
- (Akidau, https://www.oreilly.com/ideas/the-world-beyond-batch-streaming-101)

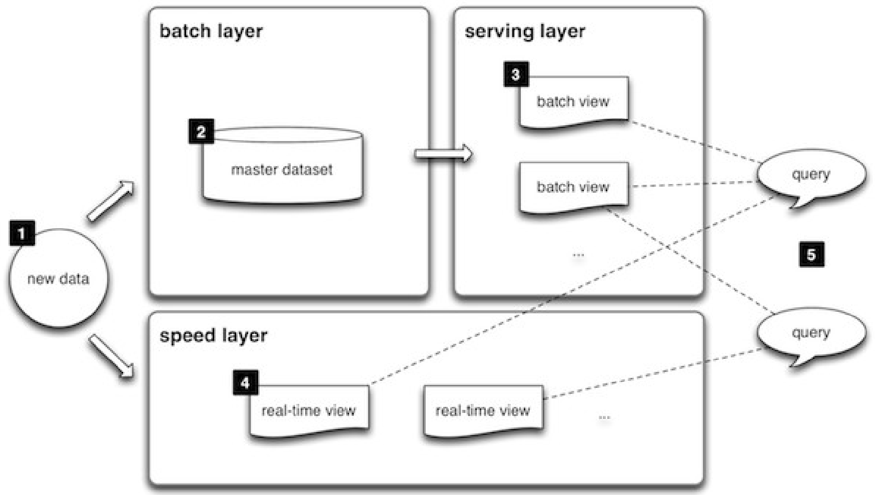


The Shortcomings of Batching

- When a dataset is complete, batch processing engines are a great way to produce an accurate and holistic view of your data.
- But what if your dataset is *not* complete? Waiting for all your data to arrive before analyzing that data via a batch job is expensive, both in terms of time and money. If your business needs NRT (near real-time) feedback on customer interaction, relying on batch processing can be a liability.
- Additionally, batch processing engines often lack the ability to effectively reason about time. This includes the notion of data that arrives out-of-order (older events arrive in a system before newer ones) or late data, events that should have been part of an earlier batch job based on the time they occurred, but were not seen until much later due to network latency or some other issue - of which there are many!

Enter Streaming Systems and the Lambda Architecture

- Stream processing engines such as Apache Beam, Apache Storm, and others, attempt to provide low-latency feedback on near real-time data. But effective stream processing engines are still relatively young. As a result, streaming systems were great for producing a view of a dataset quickly, but potentially unreliable with regard to correctness of the results they produced.
- This discrepancy gave rise to the *Lambda Architecture,* which consists of running a stream processing system side by side with periodic runs of a batching engine, and reconciling the two datasets as results are produce from both.



A Brief Note On Terminology

- The “Lambda” Architecture should not be confused with Amazon Web Services Lambda (an event-driven, serverless computing platform), or lambda functions (potentially anonymous subroutines).
- If you are interested in AWS Lambdas, checkout *Introduction to Developing AWS Lambdas in Java* by JavaMN’s own Richard Monson-Haefel: https://www.linkedin.com/in/monsonhaefel/
- If you are interested in Java Lambda Expressions, please see the Java Trail for Classes and Objects: https://docs.oracle.com/javase/tutorial/java/javaOO/index.html

Problems With the Lambda Architecture

- The lambda architecture requires two different code-bases to be maintained.
- Two different code bases means ***at least* twice** the opportunity for bugs and inconsistencies to arise (at least in my experience).
- There is an inherent amount of distrust associated with the streaming systems engine, which may mean that the streaming architecture is more trouble than its worth.

Apache Beam, The Unified Model

- Beam provides a way to get the best of both batch and stream processing without running two different systems along side one another. With Beam you can facilitate stream processing, run batch jobs, or combine some measure of both simultaneously.
- The beam framework provides SDKs (software development kits) for Java, Python, and Golang. Naturally, this presentation will focus on the Java SDK.
- Much like SLF4J (Simple Logging Facade for Java) beam gives developers a common language that may fluently interchange a variety of “runners” or “back-end” processing engines: these include Apache Apex, Apache Flink, Google Cloud Dataflow, Apache Samza, and the like.

A Brief History of Apache Beam

- Beam began at Google as ***Google Cloud Dataflow*** and was released to the public in 2015. The Dataflow whitepaper is available online, and also included as part of the GitHub repository where this slide deck is stored.
- In 2016 the the SDK was open-sourced to the Apache Software Foundation where it incubated before becoming a top level project on January 10th, 2017 (https://beam.apache.org/blog/2017/01/10/beam-graduates.html).
- Beam incorporates attributes of other projects such as ***MapReduce***, ***FlumeJava***, and ***MillWheel***, all originating at Google.

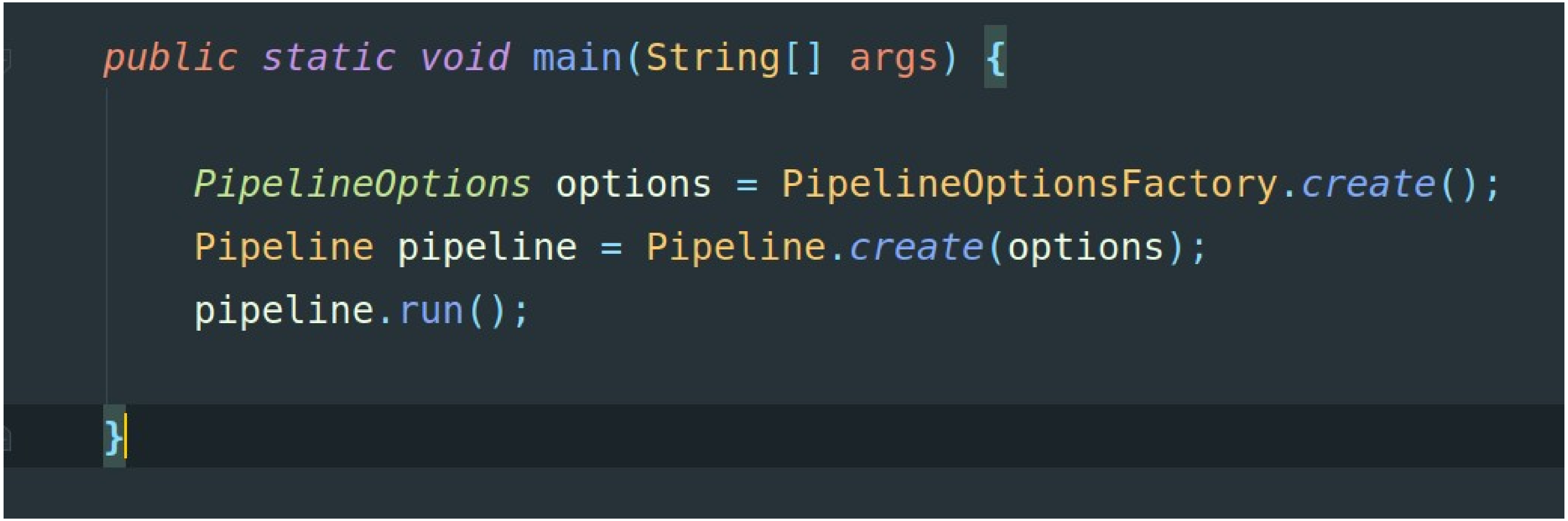
Beam in Practice – Core Features

- Pipelines: **a P*ipeline*** defines the actions taken to process data.
- PCollections: ***a PCollection*** (a contraction of parallel collection) is a potentially unbounded, potentially distributed, immutable, dataset. Your pipeline processes zero, one, or more PCollections.
- PTransforms: ***a PTransform*** (a contraction of parallel transform) defines an operation used to manipulate a PCollection. PTransforms take a PCollection as an argument and return a new PCollection (recall their immutable nature) reflecting the operations performed on the input.

Creating A Pipeline With The Java SDK

- A Beam Pipeline can be created simply, with a *PipelineOptions* class as an argument that provides valuable runtime information such as the environment to which a pipeline should be deployed, log level specifications, or the level of horizontal or vertical scaling that is required to effectively process a requisite amount of data efficiently.
- At its core, a *Pipeline* is the primary vehicle for batch or stream processing, to which PTransforms are applied.

A Brief Look At Pipeline Creation

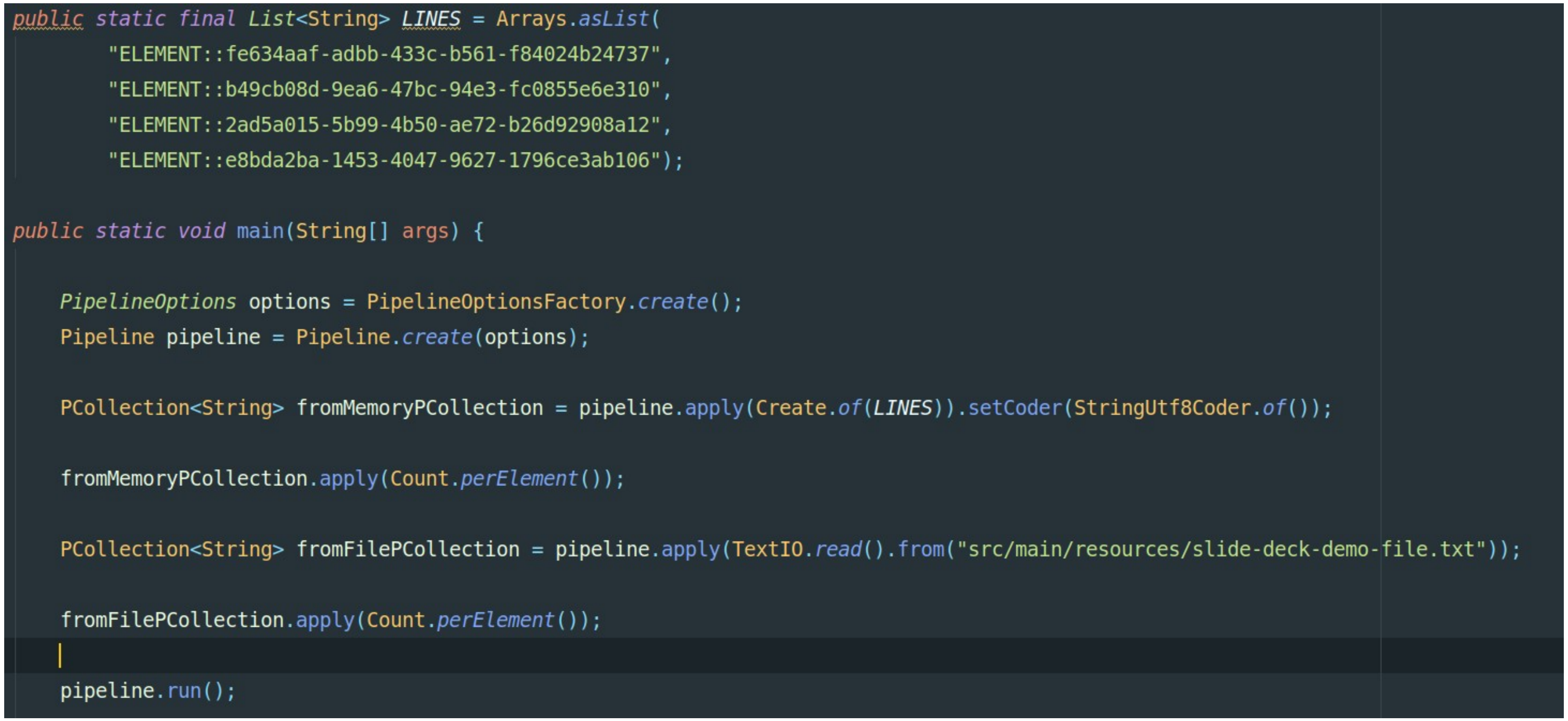


```
public static void main(String[] args) {

    PipelineOptions options = PipelineOptionsFactory.create();
    Pipeline pipeline = Pipeline.create(options);
    pipeline.run();

}
```

Creating A PCollection

- *PCollections* can be created from in memory data or read from an internal or external, bounded or unbounded datasource. The Beam SDK provides a number of pre-written *IO Adapters* to consume from common persistent data stores or messages buses.

A Brief Look At PCollection Creation



```
public static final List<String> LINES = Arrays.asList(
        "ELEMENT::fe634aaf-adbb-433c-b561-f84024b24737",
        "ELEMENT::b49cb08d-9ea6-47bc-94e3-fc0855e6e310",
        "ELEMENT::2ad5a015-5b99-4b50-ae72-b26d92908a12",
        "ELEMENT::e8bda2ba-1453-4047-9627-1796ce3ab106");

public static void main(String[] args) {

    PipelineOptions options = PipelineOptionsFactory.create();
    Pipeline pipeline = Pipeline.create(options);

    PCollection<String> fromMemoryPCollection = pipeline.apply(Create.of(LINES)).setCoder(StringUtf8Coder.of());

    fromMemoryPCollection.apply(Count.perElement());

    PCollection<String> fromFilePCollection = pipeline.apply(TextIO.read().from("src/main/resources/slide-deck-demo-file.txt"));

    fromFilePCollection.apply(Count.perElement());

    pipeline.run();
```

Creating A PTransform

- PTransforms define the operations to be performed on a PCollection. A combination of PTransforms and the PCollections that they operate on define the structure of Beam's workflow DAG (directed acyclic graph - *a graph that is directed and without cycles connecting the other edges*).
- https://beam.apache.org/documentation/programming-guide/

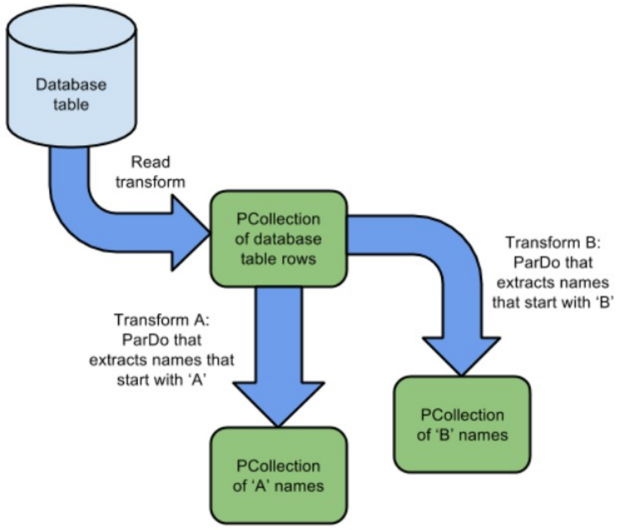
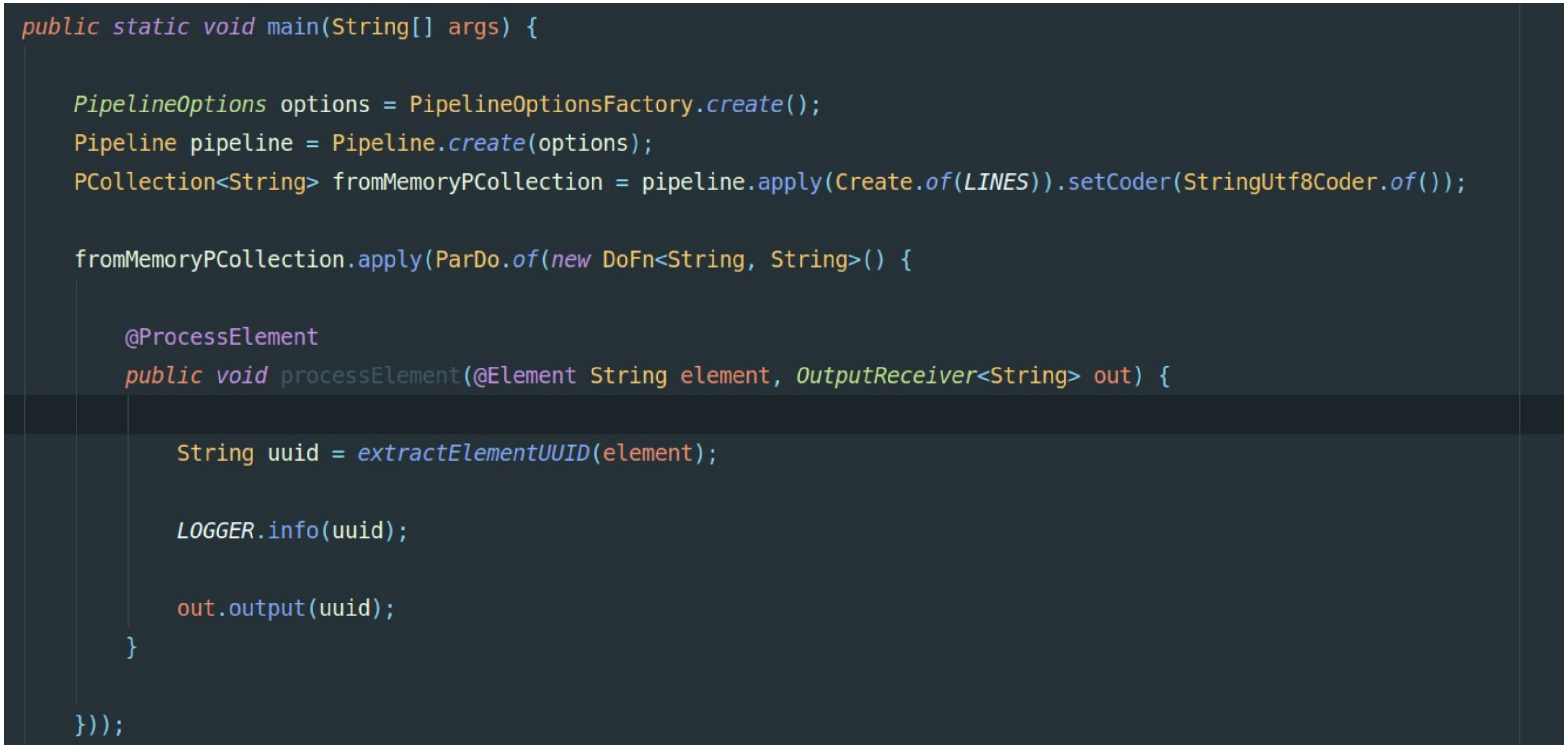

Database table
Read transform
PCollection of database table rows
Transform B: ParDo that extracts names that start with 'B'
Transform A: ParDo that extracts names that start with 'A'
PCollection of 'B' names
PCollection of 'A' names

Figure: A branching pipeline. Two transforms are applied to a single PCollection of database table rows.

Core PTransforms

- **ParDo**, similar to the *Map* phase of *Map, Filter, Reduce*, allows a developer to define the behavior of a function to be applied to each element of a PCollection.
- **GroupByKey**, takes a *PCollection*<*KV*<*K, V*>>, groups the values by key and windows, and returns a *PCollection*<*KV*<*K, Iterable*<*V*>>> representing a map from each distinct key and window of the input PCollection to an *Iterable* over all the values associated with that key in the input per window.
- **Combine**, allows a client to perform simple arithmetic operations on a data set (such as sum its contents) or to define the behavior of more complex expressions.
- **Flatten**, merge PCollections of the same type.
- **Partition**, break a part a single PCollection into smaller PCollections.

A Brief Look at PTransforms



```
public static void main(String[] args) {

    PipelineOptions options = PipelineOptionsFactory.create();
    Pipeline pipeline = Pipeline.create(options);
    PCollection<String> fromMemoryPCollection = pipeline.apply(Create.of(LINES)).setCoder(StringUtf8Coder.of());

    fromMemoryPCollection.apply(ParDo.of(new DoFn<String, String>() {

        @ProcessElement
        public void processElement(@Element String element, OutputReceiver<String> out) {

            String uuid = extractElementUUID(element);

            LOGGER.info(uuid);

            out.output(uuid);
        }

    }));
```

Serialization

- Apache Beam *Pipelines* are designed to scale horizontally, distributing *PCollections* and processing tasks across some number of worker machines. Because of this, the functions you provide to your pipeline need to be *Serializable*, and indeed, some of the core transforms already implement the Java *Serializable* Interface.

Serialization Best Practices

- “Transient fields in your function object are not transmitted to worker instances, because they are not automatically serialized.
- Avoid loading a field with a large amount of data before serialization.
- Individual instances of your function object cannot share data.
- Mutating a function object after it gets applied will have no effect.
- Take care when declaring your function object inline by using an anonymous inner class instance. In a non-static context, your inner class instance will implicitly contain a pointer to the enclosing class and that class’ state. That enclosing class will also be serialized, and thus the same considerations that apply to the function object itself also apply to this outer class.”
- https://beam.apache.org/documentation/programming-guide/#side-inputs

Thread Compatibility

- “Your function object should be thread-compatible. Each instance of your function object is accessed by a single thread at a time on a worker instance, unless you explicitly create your own threads. Note, however, that the Beam SDKs are not thread-safe. If you create your own threads in your user code, you must provide your own synchronization. Note that static members in your function object are not passed to worker instances and that multiple instances of your function may be accessed from different threads.”
- https://beam.apache.org/documentation/programming-guide/#side-inputs

Idempotency

- “It’s recommended that you make your function object idempotent–that is, that it can be repeated or retried as often as necessary without causing unintended side effects. Non-idempotent functions are supported, however the Beam model provides no guarantees as to the number of times your user code might be invoked or retried; as such, keeping your function object idempotent keeps your pipeline’s output deterministic, and your transforms’ behavior more predictable and easier to debug.”
- https://beam.apache.org/documentation/programming-guide/#side-inputs

Reasoning About Time

- Beam defines a number of advanced features for breaking up unbounded data into discrete datasets for processing.
- To understand how the beam model accomplishes this, it helps to start thinking about event driven systems in terms of two time vectors: *event time* and *processing time.*
- *Event time* denotes the moment an event actually occurred. Processing time is the moment at which that event arrives at, and is recognized by the system that intends to process the event.
- It is important to note that event time is almost always earlier than processing time (and never later), and sometimes much earlier due to network latency. Additionally, one event that occurs before another in terms of event time may arrive to a processing system *after* the event with a later event time.

Windowing

- When dealing with unbounded data, PTransforms like *GroupByKey* require some sort of bound on which to operate, since otherwise there is an infinite amount of data to key by. And indeed, a *GroupByKey* operation groups not only by key, but by a temporal *window* defined in your pipeline. We'll see an example of this shortly.
- While there are several kinds of of windowing available to developers, including *Session Windowing,* and *sliding windows*, this presentation focuses on *fixed event time windowing,* in which an unbounded dataset is periodically divided into fixed duration s

Fixed Windows Visualized

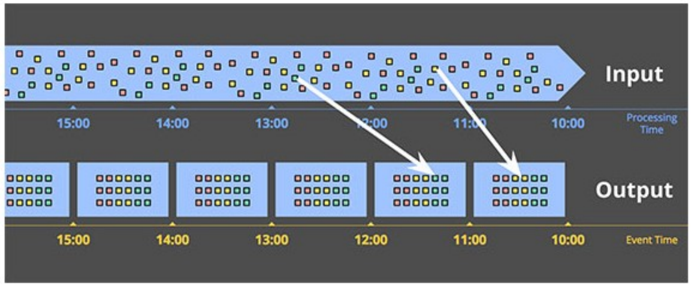


Figure 10: Windowing into fixed windows by event time. Data are collected into windows based on the times they occurred. The white arrows call out example data that arrived in processing time windows that differed from the event time windows to which they belonged. Image: Tyler Akidau.

- Fixed windows by event time means that regardless of when an element of event time *t* arrives for processing, it should still be grouped into its associated time-frame, where *windowMin* <= *t* <= *windowMax*. https://www.oreilly.com/ideas/the-world-beyond-batch-streaming-101

Event Time Skew

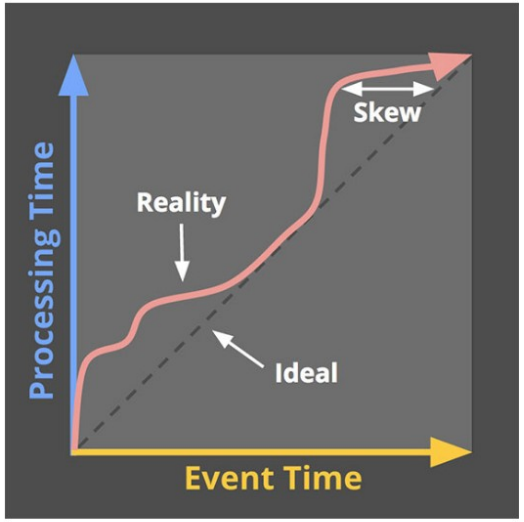


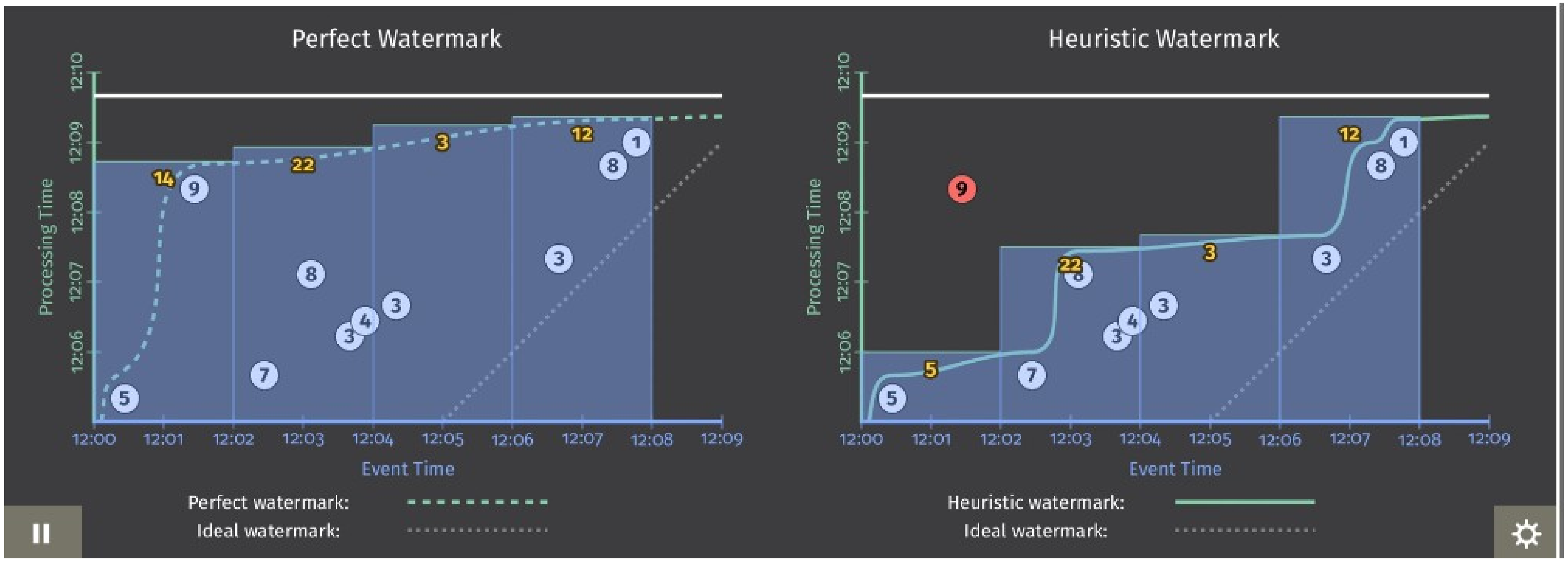
Figure 1: Example time domain mapping. The X-axis represents event time completeness in the system, i.e. the time X in event time up to which all data with event times less than X have been observed. The Y-axis represents the progress of processing time, i.e. normal clock time as observed by the data processing system as it executes. Image: Tyler Akidau.

- In a perfect world, event time might equal processing time. This graph shows a more realistic picture how of and when events are realized in a system.
- https://www.oreilly.com/ideas/the-world-beyond-batch-streaming-101

Watermarks

- Beam tracks the notion of something called a *watermark*, which closely tracks the event time skew displayed on the previous slide.
- In real time streaming scenarios, and when reading from an unbounded data source, Beam uses the watermark heuristic to help estimate when the pipeline, or any particular state of the pipeline, has seen all events with event times that precede some time *t*.

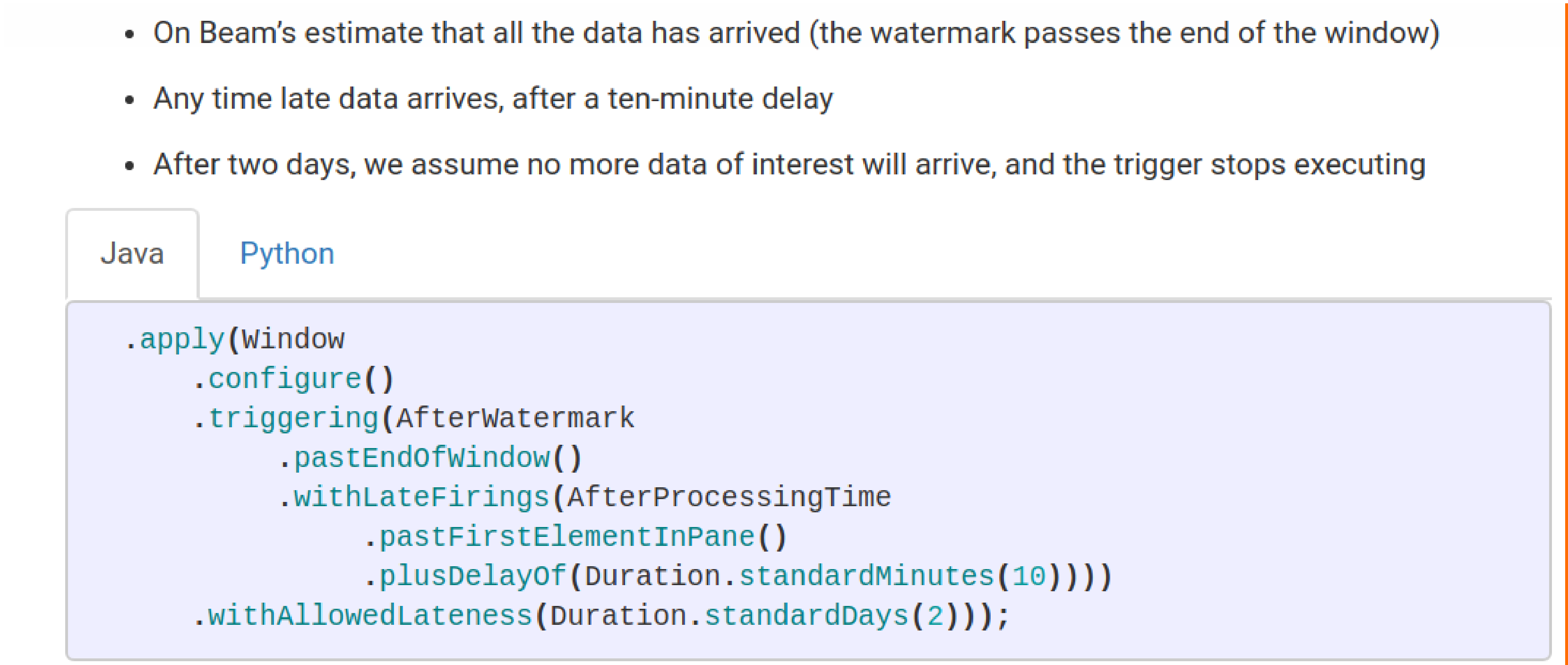
Watermarks Visualized



Triggering And Pane Emission

- The notion of the watermark gives Beam Programmers the ability to decide *when* to emit the results of their windowed data, in addition to how to handle late arriving (and even early!) data.
- In combination, windowing and triggering provide the sophisticated mechanisms for handling near real-time, unbounded data, that makes Apache a great stream processing tool.
- The fact that these APIs can also be used for simple (or complex) batch processing jobs is what the Beam Docs say when they define the framework as a *Unified Model*.

A Brief Look At Window and Trigger Creation



- On Beam's estimate that all the data has arrived (the watermark passes the end of the window)
- Any time late data arrives, after a ten-minute delay
- After two days, we assume no more data of interest will arrive, and the trigger stops executing

```java
  .apply(Window
      .configure()
      .triggering(AfterWatermark
           .pastEndOfWindow()
           .withLateFirings(AfterProcessingTime
                .pastFirstElementInPane()
                .plusDelayOf(Duration.standardMinutes(10))))
      .withAllowedLateness(Duration.standardDays(2)));
```

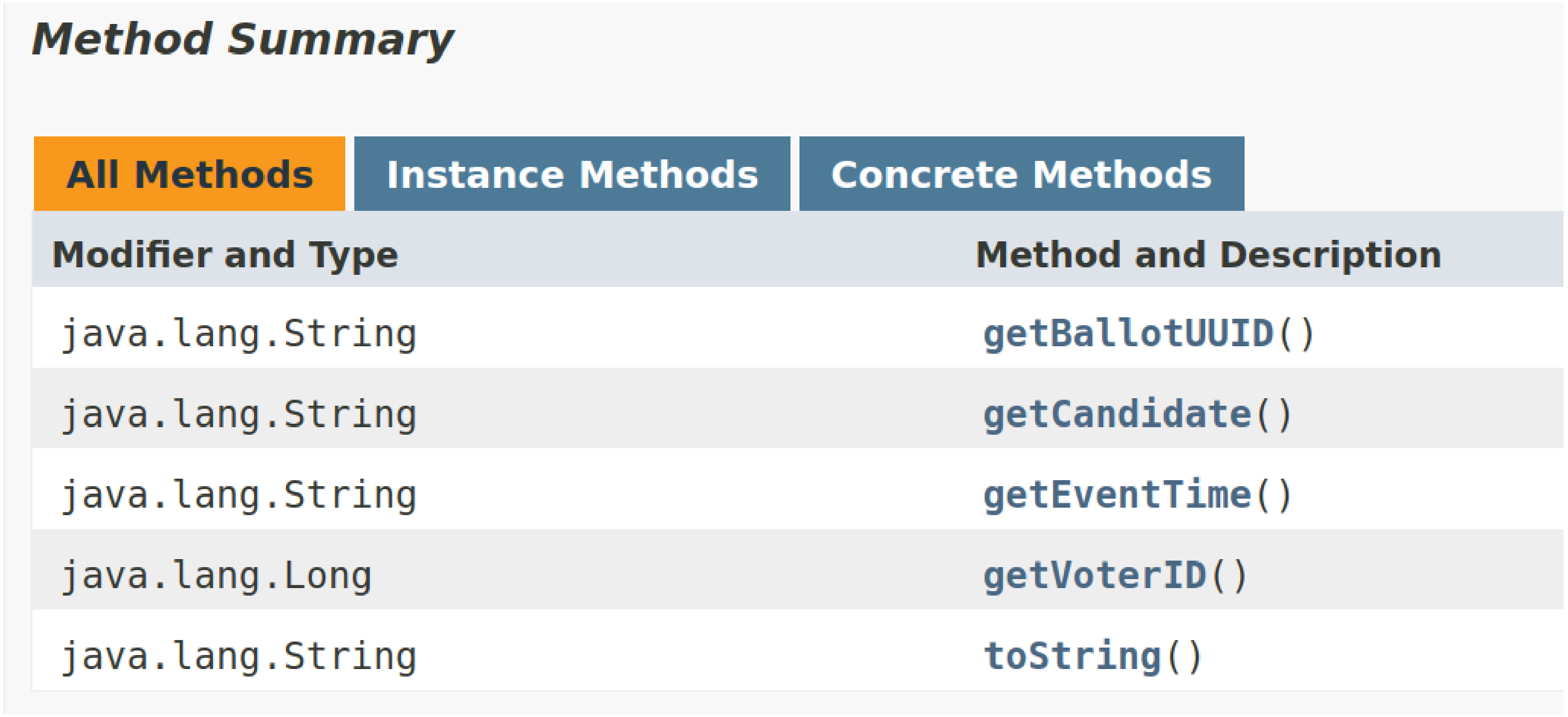
Case Study: Election Night Analysis

- Consider the challenge of efficiently counting millions of ballots, identifying fraudulent votes, and producing near real-time data for news coverage over the course of an election day.
- Apache Beam provides the tools to do all of this through both batch and streaming paradigms. For the purposes of this presentation we have a hypothetical pool of bounded data (over 1000 ballots collect) we will analyze using the Beam API.

The Ballot Domain Object

- Ballots consist of a unique Voter ID, a Ballot UUID, a Ballot timestamp that is produced when the ballot is fed into the reader, and naturally a candidate that the voter has chosen to support.
- In our model, a Ballot is fraudulent if either the Voter ID or the Ballot UUID is repeated.

A Brief Look at Ballot API



Method Summary

Modifier and Type	Method and Description
`java.lang.String`	`getBallotUUID()`
`java.lang.String`	`getCandidate()`
`java.lang.String`	`getEventTime()`
`java.lang.Long`	`getVoterID()`
`java.lang.String`	`toString()`

Additional Resources

- https://beam.apache.org/
- https://www.oreilly.com/ideas/the-world-beyond-batch-streaming-101
- https://www.oreilly.com/ideas/the-world-beyond-batch-streaming-102
- http://streamingsystems.net/